Automatic Detection of Inter-application Permission Leaks in Android Applications

Technical Report TR13-02, Department of Computer Science, Rice University, January 2013

Dragoş Sbîrlea
Rice University
dragos@rice.edu

Michael G. Burke
Rice University
mgb2@rice.edu

Salvatore Guarnieri
IBM Watson Research Center
sguarni@us.ibm.com

Marco Pistoia
IBM Watson Research Center
pistoia@us.ibm.com

Vivek Sarkar
Rice University
vsarkar@rice.edu

ABSTRACT

Due to their growing prevalence, smartphones can access an increasing amount of sensitive user information. To better protect this information, modern mobile operating systems provide permission-based security, which restricts applications to only access a clearly defined subset of system APIs and user data. The Android operating system builds upon already successful permission systems, but complements them by allowing application components to be reused within and across applications through a single communication mechanism, called the `Intent` mechanism.

In this paper we identify three types of inter-application `Intent`-based attacks that rely on information flows in applications to obtain unauthorized access to permission-protected information. Two of these attacks are of previously known types: confused deputy and permission collusion attacks. The third attack, private activity invocation, is new and relies on the existence of difficult-to-detect misconfigurations introduced because `Intent`s can be used for both intra-application and inter-application communication. Such misconfigured applications allow protected information meant for intra-application communication to leak into unauthorized applications. This breaks a fundamental security guarantee of permissions systems: that applications can only access information if they own the corresponding permission.

We formulate the detection of the vulnerabilities on which these attacks rely as a static taint propagation problem based on rules. We show that the rules describing the permission-protected information can be automatically generated though static analysis of the Android libraries - an improvement over previous work. To test our approach we built PermissionFlow, a tool that can reliably and accurately identify the presence of vulnerable information flows in Android applications.

Our automated analysis of popular applications found that 56% of the top 313 Android applications actively use inter-component information flows; by ensuring the absence of inter-application permission leaks, the proposed analysis would be highly beneficial to the Android ecosystem. Of the tested applications, PermissionFlow found four exploitable vulnerabilities.

1. INTRODUCTION

Users of modern smarphones can install third-party applications from markets that host hundreds of thousands of applications [1, 2] and even more from outside of official markets. To protect sensitive user information from these potentially malicious applications, most operating systems use permission-based access-control models (Android [3], Windows Phone 7 [4], Meego [5] and Symbian [6]).

Permissions are a well known and powerful security mechanism, but - as with any new operating system - there is the possibility that Android-specific features may reduce the guarantees of the classic permissions model. One such feature is the new communication mechanism (called `Intent`s), which can be used to exchange information between components (called `Activity`s) of the same application or of different applications.

One attack that exploits `Intent`s for malicious purposes is *permission collusion*. In this attack, an application that individually only has access to harmless permissions augments its capabilities by invoking a collaborating application through sending and receiving `Intent`s. To stage this attack, malevolent developers could trick users into installing such cooperating malicious applications that covertly compromise privacy.

A second type of attack using `Intent`s is the *confused deputy attack*. Confused deputy attacks rely on misconfigured applications; components that interact with other applications are invoked by unauthorized callers and allow them to perform protected actions or access permission-protected information.

We discovered a third type of attack, *private activity invocation*, which is a more virulent form of the confused deputy attack: it affects applications not meant to communicate

with other applications. Even if a developer's intention was to disallow external invocation of internal `Activity`s, other applications may be able to invoke them if the application does not have the necessary configuration. This is possible because `Intent`s can be used for inter-application invocations as well as intra-application invocations.

One possible approach for leveraging static analysis to discover these vulnerabilities is to merge the call graph of each application and to enhance it with edges representing the possible call edges between applications. On this inter-application call graph, one could check using static taint analysis whether protected information leaks to applications that do not own the required permissions. However, this approach could be time-consuming because of the large number of applications that must be analyzed together.

We propose an alternative approach that first summarizes the permission-protected APIs of the Android libraries. Then, using taint analysis, it tracks the information flow through both the Android libraries and the application, checking if the information reaches a misconfigured or otherwise vulnerable component that allows permission-protected information to escape to other applications.

Tested on 313 popular Android Market applications, our tool, PermissionFlow, identified that 56% of them use inter-component information flows that may require permissions. Four exploitable vulnerabilities were found. The structure of the paper is the following. The relevant parts of the Android development model and permission system are described in Section 2. Section 3 illustrates possible attack scenarios.

We express the vulnerabilities as a taint propagation problem in Section 4 and in Section 5 we present the design of our analysis. Its main components are the Permission Mapper, which summarizes the protected information sources accessible to any application, and the Rule Generator, which generates taint rules that specify vulnerable flows; these rules serve as input for our static analysis engine. We experimentally evaluate our analysis in Section 6. We further discuss our experimental findings, and make recommendations for Android application security, in Section 7. Section 8 is dedicated to related work, and we conclude in Section 9. Our contributions are the following:

- We classify `Intent`-based vulnerabilities in Android applications, identifying a new variant of the confused deputy attack, and describe ways of protecting against attacks.
- We developed PermissionFlow, a tool for discovering vulnerabilities in the bytecode and configuration of Android applications.
- We propose a novel approach for automatic generation of taint flow sources for permission-based systems by considering permission-protected APIs to be sources of taint. Our Permission Mapper improves on previous work by performing fully automatic analysis for Android Java APIs.
- We evaluate PermissionFlow on leading Android Market applications and show that 177 out of the 313 applications tested would benefit from our tool. PermissionFlow found three permission-protected leaks in widely used applications and an additional vulnerability that allows leaking of information that should be protected by custom permissions.

2. BACKGROUND

The vulnerabilities we identify involve knowledge about the Android development model, the Android inter-process communication mechanism and its permissions system. These components are the focus of the following subsections.

2.1 Android development

Android applications are typically written in Java using both standard Java libraries and Android-specific libraries. On Android devices, the Java code does not run on a standard JVM, but is compiled to a different register-based set of bytecode instructions and executed on a custom virtual machine (Dalvik VM). Android application packages, also called *APK*s after their file extension, are actually ZIP archives containing the Dalvik bytecode compiled classes, their associated resources such as images and the application manifest file.

The application manifest is an XML configuration file (AndroidManifest.xml) used to declare the various components of an application, their encapsulation (public or private) and the permissions required by each of them.

Android APIs offer programmatic access to mobile device-specific features such as the GPS, vibrator, address book, data connection, calling, SMS, camera, etc. These APIs are usually protected by permissions.

Let's take for example the `Vibrator` class: to use the `android.os.Vibrator.vibrate(long milliseconds)` function, which starts the phone vibrator for a number of milliseconds, the permission `android.permission.VIBRATE` must be declared in the application manifest, as seen on line 2 of Listing 1.

Application signing is a prerequisite for inclusion in the official Android Market. Most developers use self-signed certificates that they can generate themselves, which do not imply any validation of the identity of the developer. Instead, they enable seamless updates to applications and enable data reuse among sibling applications created by the same developer. Sibling applications are defined by adding a `sharedUid` attribute in the application manifest of both, as seen in line 1 of Listing 1.

`Activity`s. The Android libraries include a set of GUI components specifically built for the interfaces of mobile devices, which have small screens and low power consumption. One type of such component is `Activity`s, which are windows on which all visual elements reside. An `Activity` can be a list of contacts from which the user can select one, or the camera preview screen from which he or she can take a picture, the browser window, etc.

`Intent`s. Applications often need to display new `Activity`s . For example, choosing the recipient of an SMS message is performed by clicking on a button that spawns a new `Activity`.

```
<manifest package="com.android.app.myapp" sharedUid="uidIdentifier">
    <uses-permission  name="android.permission.VIBRATE" />
    <activity name="MyActivity">
          <intent-filter>
              <action name="com.zxing.SCAN" />
              <category name="category.DEFAULT" />
          </intent-filter>
    </activity>
</manifest>
```

Listing 1: An `Activity` declaration in AndroidManifest.xml with declarations of used permissions and an intent filter.

This `Activity` displays the contacts list and allows the user to select one. To spawn the new `Activity`, the programmer creates a new `Intent`, specifies the name of the target class, and then starts it, as shown in the following snippet:

```
Intent i = new Intent();
i.setClassName( this, "package.CalleeActivity" );
startActivity(i);
```

Usually the parent `Activity` needs to receive data from the child `Activity`, such as - in our SMS example above - the contact phone number. This is possible through the use of `Intent`s with return values, The parent spawns a child by using `startActivityForResult()` instead of `startActivity()` and is notified when the child returns through a callback (the `onActivityResult()` function), as shown in Listing 2. This allows the parent to read the return code and any additional data returned by the child `Activity`.

```
void onActivityResult( int requestCode, int
     resultCode, Intent data) {
   if (requestCode == CREATE_REQUEST_CODE)
       {
     if (resultCode == RESULT_OK) {
       String info = intent.getStringExtra(
           "key");
     }
   }
}
```

Listing 2: Code snipped showing how a caller accesses information returned by a child `Activity`.

```
Intent intent = new Intent();
intent.putExtra("key", "my value");
this.setResult(RESULT_OK, intent);
finish();
```

Listing 3: Code snippet showing how child `Activity`s can return data to their caller.

As shown in Figure 3, the child `Activity` needs to call the `setResult` function, specifying its return status. If additional data should be returned to the parent, the child can attach an `Intent` along with the result code and supply extra key/value pairs, where the keys are Java `String`s and the values are instances of `Parcelable` types, which are similar to Java `Serializable` classes and include `String`s, arrays and value types.

Sending `Intent`s to explicitly named `Activity`s , as described above, is called explicit `Intent` usage. Android also allows creation of `Intent`s specifying a triple (action, data type, category) and any `Activity` registered to receive those attributes through an `intent-filter` will be able to receive of the `Intent`. If there are multiple `Activity`s that can receive the `Intent`, the user will be asked to select one.

The explicit `Intent` feature is mostly used in intra-application communication, as described in the following section, but can be useful for inter-application communication too and its existence is the root cause of the vulnerabilities discovered by us.

2.2 Inter-application Intents and data security

Inter-process communication with `Intent`s. `Intent`s can be used for communication between `Activity`s of the same application or for inter-application communication. In the second case, `Intent`s are actually inter-process message-passing primitives. To specify a subset of `Intent`s that an `Activity` answers to, developers add to the application manifest an `intent-filter` associated with the `Activity`. The `intent-filter` in Figure 1 specifies that `MyActivity` can be invoked by sending an `Intent` with action `com.zxing.SCAN`; such an `Intent` is called an implicit `Intent` because it does not specify a particular `Activity` to be invoked. Implicit `Intent`s are created using the single parameter constructor `new Intent(String)`.

Component encapsulation. Developers enable or disable inter-application invocation of their `Activity`s by setting the value of the boolean `exported` attribute of each `Activity` in the application manifest. The behavior of this attribute is a detail that may be a source of confusion, as the meaning depends on the presence of another XML element, the `intent-filter`:

- If an `intent-filter` is declared and the `exported` attribute is not explicitly set to `true` or `false`, its default value is `true`, which makes the `Activity` accessible by any application.
- If an `intent-filter` is not declared and the `exported` attribute is not set, by default the `Activity` is only accessible through `Intent`s whose source is the same application.

An exception to the above rules is allowed if the developer

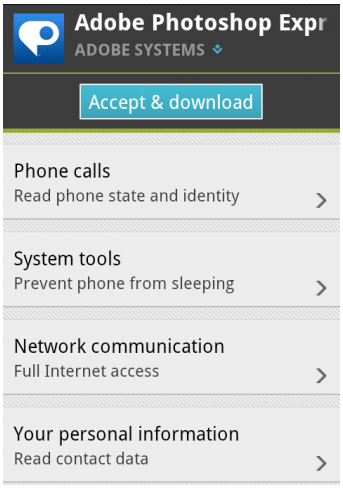

Figure 1: Before installing any application, the user is presented with a list of permissions that the application needs access to.

specifies the attribute `sharedUid` in the manifest file. In that case, another application may run in the same process and with the same Linux user ID as the current application. This addition changes the behavior of `Activity`s that are not exported: they can be invoked not only from the same application, but also from the sibling application with the same user ID. Listing 1 shows the use of the `sharedUserId` attribute.

It is important to realize that the `intent-filter` mechanism does not provide any security guarantees and is meant only as a loose binding between `Activity`s and `Intent`s; any `Activity` with an `intent-filter` can still be sent an explicit `Intent` in which case the `intent-filter` is ignored. The presence of this attribute, however, changes the behavior of the security-related `exported` attribute, as detailed above. We found that many developers overlook the security-related implications when using `intent-filter`s.

2.3 Android permissions system

For a user, Android permissions are just lists of capabilities that he or she has to accept before installing applications.

As seen in Figure 1, when installing an application from the official Android Market, the user is presented with a list of permission names, each with a short description.

From the point of view of the Android programmer, each permission provides access to one or more Android Java APIs that would otherwise throw an exception when used. Permissions also protect Android `ContentProvider`s, which are SQLite databases indexed using a URI. Different URIs mean different permissions might be needed to access the corresponding data.

To request permissions, the developer needs to declare them in the application manifest, as seen in Listing 1, through an `uses-permission` attribute that specifies the exact permission as a `String` value. Users have a reasonable expectation that if they do not give permission to an application to access information (for example, their contacts), that application will not have access to that information through some other means.

3. ATTACKS

All Android applications with a graphical user interface contain at least one `Activity`, which means vulnerabilities related to `Activity`s can affect a majority of applications. All the vulnerabilities that we identify have in common the existence of information flows that are meant to allow child `Activity`s to communicate with authorized parents, but can instead be used by unauthorized applications to access sensitive information without explicitly declaring the corresponding permission.

We identified three different attack scenarios, discussed in the following paragraphs and out tool, PermissionFlow, identifies the flow vulnerabilities that enable all of them. PermissionFlow validation can be used as a prerequisite for applications before being listed in Android Market and by developers to ensure the security of their applications or by users.

Note that other operating systems sandbox applications and do not offer a mechanism for direct inter-application communication; in spite of this, some of these attacks are still possible. For example, in iOS, the colluding applications attack can be performed through URL Schemes [7].

1. Attacks on misconfigured applications happen when an attacker application installed on the device can exploit the flows of a misconfigured application. If an application has any one of the configuration parameter combinations listed in Table 1 as high risk, then any application on the device can spawn it.

If in the application manifest an `Activity` is listed with an `intent-filter` and is not accompanied by a `exported="false"` attribute, any other application on the system can invoke it. Then, in the absence of dynamic permission checking by the developer (not standard practice), information returned to the caller through the `Intent` result may compromise permission-protected information, as no permission is required of the caller.

In the example from Figure 2 (left), the user installed malevolent application B (a music streaming app) with permission to access the Internet. B can exploit the honest but misconfigured contact manager application A by invoking Activity A2 that returns the contacts; B can then send the contacts to a remote server. If A2 is built to reply to external requests and it just failed to check that B has the proper permission, then the attack is a classic **confused deputy** attack. However, because in Android `Intent`s are also used as an internal (intra-application) communication mechanism, it is possible that A2 is not built for communicating with another application and is just misconfigured. This **private activity invocation** is a more powerful attack than confused deputy because it targets internal APIs and not only public entry points. These internal APIs are generally not regarded as vulnerable to confused deputy and so not they are not secured against it. Further, by increasing the number of APIs that can be targeted, this attack increases the likelihood that the returned information is permission-protected (Protected information tends to flow between internal components such

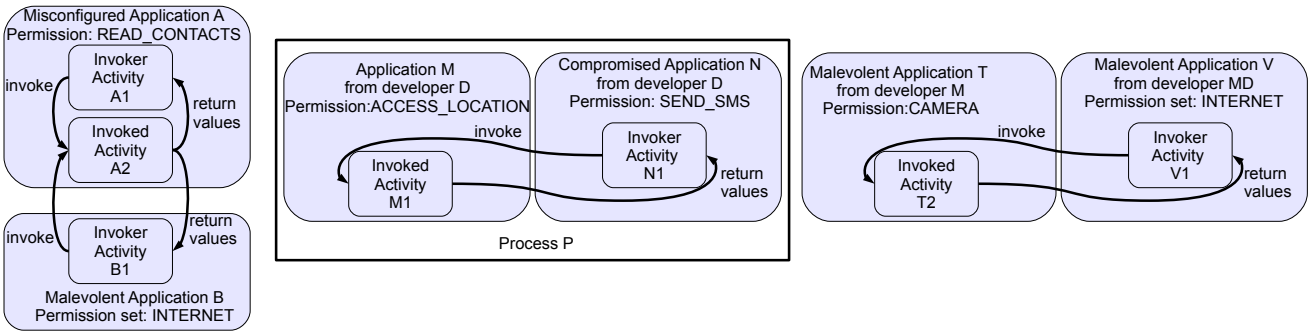

Figure 2: Possible attacks: internal `Activity` invocation or confused deputy, application sharing user ID with a compromised application (center) and permission collusion by malevolent applications (right).

as A1 and A2, even when it does not leave the application.) PermissionFlow allows developers to identify the existence of permission-protected information flowing between components and use this information to properly configure their applications.

For `Activity`s that are designed to be invoked by unknown applications, developers can ensure that callers own a set of permissions in one of two ways: declaratively (in the manifest file, using the `permission` attribute of the `Activity`) or dynamically (by calling the function `checkCallingPermission( String permission)`). Note that the `permission` attribute can only be used to enforce a single permission and is different from the `uses-permission` node in Listing 1, which controls what permissions the application needs in order to function.

The safest approach is to completely disable outside access to internal `Activity`s that may leak protected information. Table 1 shows the combinations of configuration parameters that may lead to information leaks. Each of the combinations also lists if any callers are allowed for that `Activity` or if the `Activity` restricts access to only applications from the same developer.

2. Collusion attacks obtain permission-protected information without requesting the permission, by exploiting the combination of assignment of Android permissions on a per-application basis and the exchange of applications information without making this explicit to the user.

In Figure 2 (right), we show a scenario in which a user is tricked by a malevolent developer MD into installing two separate applications, that seem to have little risk associated with them. For example, a camera application that does not require the `Internet` permission seems safe, as it cannot upload the pictures to the Internet. Similarly, a music streaming application that does not request the `Camera` permission would be acceptable. However, if the two applications are malicious, the music streaming application can invoke the camera application and send the pictures obtained from it remotely. The Android security system does not inform the user of this application collusion risk.

Note that the camera application can include checks on the identity of the caller, such that it returns the pictures to no other application except the music streaming one, which allows such colluding applications to pass a dynamic security analysis that invokes all possible `Activity`s and checks the returned information.

3. Attack on applications sharing the user ID. We have not yet discussed an additional type of attack that our approach can recognize, but is improbable in practice because of its narrow applicability. This type of attack is on sibling applications.

These attacks target honest developers who had one of their applications compromised through methods unrelated to our work, such as:

- The developer's certificate is compromised (these are self-signed certificates, vulnerable to exploitation since they cannot be revoked [**?**]); the certificate can be used to sign and publish a malevolent update to an application N of that developer.
- One of the developer's applications N is compromised directly through some other vulnerability.

This vulnerability allows an attacker to access the permission-protected information of applications sharing the user ID with the already compromised N. If N is configured to have the same user ID as application M (as shown in Figure 2 center), it can then obtain the information from M. To set up this attack, an attacker would need to control application N of developer D, N should have any exfiltration permission (sending short messages, accessing the Internet, etc.) and N should share user ID with an application by D that returns permission-protected information. Then, N can invoke most `Activity`s of M, even if M is configured according to the rows with low risk level in Table 1. PermissionFlow can detect this vulnerability too.

4. TAINT PROPAGATION

We express the problem of leaking permission-protected information to other applications by tracking flows of sensitive information (taints) inside each application from information sources protected by permissions to values that these applications return to callers.

The taint analysis uses the following sources and sinks:

Activity configuration		Application configuration	Consequence	
Exported	**Intent-filter**	**SharedUid**	**Callers accepted**	**Risk level**
exported="true"	present	any	any	HIGH
exported="true"	absent	any	any	HIGH
exported="false"	present	set	from same developer	LOW
exported="false"	absent	set	from same developer	LOW
default	present	any	any	HIGH
default	absent	set	from same developer	LOW

Table 1: Different configurations lead to different levels of vulnerability.

- Sources: the permission-protected APIs in the Android libraries. Additionally, there may be other sources, such as callbacks registered by the application to be called when some events occur, for example when a picture is taken using the phone camera.

- Sink: `Activity.setResult(int code, Intent intent)`. The `Intent` parameter of calls to this function is accessible to the caller of the current `Activity`, so any data attached needs to be protected.

From a confidentiality perspective, applications must also protect other types of sources that are not protected by any of the standard permissions defined by Android, for example credit card numbers, account information, etc. This is done by using custom permissions that protect the invocation of their `Activity`s. The relationship between the custom permission and the data or API it protects is not obvious, so there is no way to automatically generate taint rules checking for custom permissions. PermissionFlow can track these flows only through additional rules that apply to application-specific sources of taint.

5. SYSTEM DESCRIPTION

PermissionFlow has two main parts. The first one is a general, reusable taint analysis framework; the second consists of all other components, which are Android-specific.

To analyze real Android Market applications, whose source code is usually not available, we support input in the form of Android binary application packages (APK files). This means PermissionFlow can also be used by Android users, developers and security professionals.

The system design (Figure 3) consists of the following components:

- The **Permission Mapper** (labeled 1 in the Figure 3) builds a list of method calls in the Android API that require the caller to own permissions. Its inputs are Android classes obtained by building the Android source code, with any modifications or additions performed by the device manufacturer. Having the complete system code as input allows the mapper to extract all the permissions-protected APIs that will be present on the device. It builds a permissions map, which maps permission-protected methods to their required permissions.

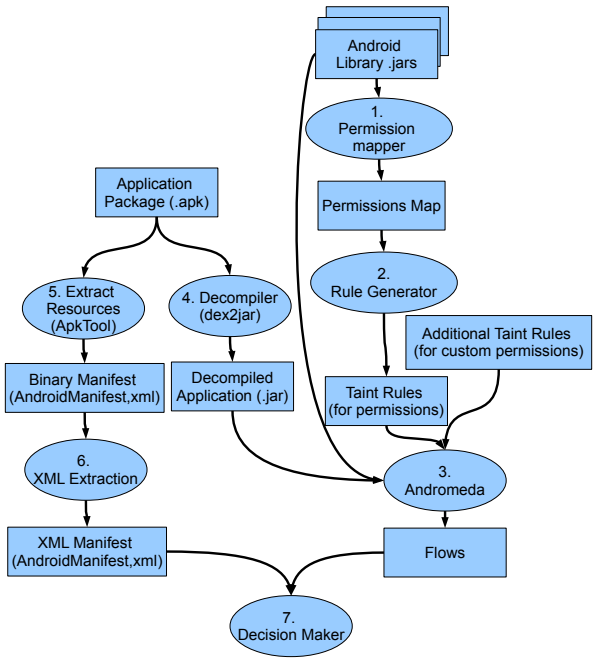

Figure 3: The components of PermissionFlow

- The permissions map is passed to the **Rule Generator**, which builds the taint analysis rules relating the sources in the map with their corresponding sinks. In our case, the only sink is the `Activity.setResult` method with an `Intent` parameter.

- Our **taint analysis engine**, (labeled 3) reads the generated rules and any extra rules manually added for detecting application-dependent private information. It outputs the flows that take the protected information from sources to sinks. For this it needs access to the application classes and the Android library classes. The taint analysis engine also needs access to the Android library, in order to track flows that go through it, for example callbacks that get registered, `Intent`s that get passed to the system, etc.

- The **dex2jar decompiler** [8] (labeled 4) is used to extract from the application APK a JAR archive containing the application bytecode.

- To extract the binary application manifest from the application package we use the **ApkTool** [9] (labeled

5); the decompilation step needed to get the textual XML representation is performed by **AXMLPrinter2** [10] (labeled 6).

- The taint analysis engine outputs the flows from sources to sinks if there are any, but the presence of flows does not in itself imply that the application is vulnerable. The **Decision Maker** (labeled 7) looks for the patterns identified in Table 1 in the application manifest file - these patterns correspond to misconfigurations that allows successful attacks to take place. If an application contains a vulnerable information flow and is improperly configured, only then is it vulnerable.

Currently, all the described components are implemented, except the decision maker, whose role was performed through manual inspection.

5.1 The Permission Mapper

The Permission Mapper matches function calls used for permission enforcement in the Android libraries to Android library functions that use these calls. In short, it uses static analysis to identify permission-protected methods and to map them to their required permissions; this analysis is independent of any application analysis and needs to be performed only once for each input Android configuration.

Identifying sources of permission-protected information is challenging because of a phenomenon known as the Android version and capability fragmentation. Android has undergone a quick succession of 15 API improvements, some with multiple revisions, most of which are still in active use today. Relying on the documentation to find which APIs require permissions would bind our analysis to a particular version of Android whose documentation we used as input. Even more differences between Android APIs are introduced by hardware manufacturers such as Samsung and HTC who build their own additions to Android (Sense and Touchwiz, respectively). These add-ons include everything from drivers and libraries to user interface skins and new system applications, which leads to capability fragmentation. Because of fragmentation, when an application performs a call to a library that is not distributed with the application, Android fragmentation makes identifying which permissions are needed for that call very difficult, as the exact permissions may be different depending on the exact Android version and add-ons. Identifying sources of permission-protected information could also be attempted by crawling the documentation. However, it is incomplete even for public, documented classes and does not include public, but non-documented methods and does not account for Java reflection on non-public methods. It also does not account for any modifications and additions to the Android API performed by the phone manufacturer. Previous work [11] showed that it is possible to identify which API calls require permissions though a combination of automated testing and manual analysis, but they use techniques that allow false negatives, need partial manual analysis and do not handle the version and feature fragmentation problem of Android.

For these reasons, we built the Permissions Mapper, a reliable and automatic tool for identifying permission-protected APIs and their required permissions. The Permissions Mapper takes as input the JAR archives of the Android distribution that needs to be summarized, including any additional code added by the hardware manufacturer. This allows for a complete analysis that works without user input and can reliably deal with API differences between various versions of the OS.

```
1  public void vibrate(long time) {
2    if (mService == null) {
3      Log.w(TAG, "Service not found.");
4      return;
5    }
6    try {
7      mService.vibrate(time, mToken);
8    } catch (RemoteException e) {
9      Log.w(TAG, "Failed to vibrate.", e);
10   }
11 }
```

Listing 4: Code snippet showing how API calls use services to perform protected functionality.

In the Android libraries, several mechanisms are used to enforce permissions:

- Calls to the `checkPermission` function located in the `Context and PackageManager` classes or `checkCallingOrSelfPermission` function located in the `Context` class;
- Linux users and groups (used, for example when enforcing the `WRITE_EXTERNAL_STORAGE` and `BLUETOOTH` permissions);
- From native code (such as the `RECORD_AUDIO` or `CAMERA` permissions).

Our work targets complete coverage of APIs enforced through the first category, which includes the majority of Android permissions.

To illustrate how permission checks work, we can use for example the `VIBRATE` permission. To use the phone vibrator, an application needs to own the `VIBRATE` permission; all functions that require this permission check for it. One such function is `Vibrator.vibrate`, whose source code is shown in Figure 4. When this function is called, the Android API forwards the call to a system service instance `mService` (line 8), which executes in a different process from the application. The `mService` instance is returned by a stub of the vibrator service:

```
mService = IVibratorService.Stub.asInterface(
    ServiceManager.getService(``vibrator") );
```

This is because in Android, developers build Android Interface Definition Language (AIDL) interfaces, from which remote invocation stubs are automatically generated [**?**], similar to Java RMI development.

The service process is the one that makes the actual permission checks, before performing any protected operation, as

```
1  public class VibratorService extends IVibratorService.Stub {
2    public void vibrate(long milliseconds, binder) {
3      if (context.checkCallingOrSelfPermission(VIBRATE)) != PackageManager.PERMISSION_GRANTED){
4              throw new SecurityException("Requires VIBRATE permission");
5      }
6    }
7  }
8
9  public class ContextImple extends Context {
10  public int checkCallingOrSelfPermission(String permission) {
11    return checkPermission(permission, Binder.getCallingPid(), Binder.getCallingUid());
12   }
13 }
```

Listing 5: Code snippet showing how services check the permissions of the application.

shown in Listing 5.

The interprocedural dataflow analysis is built using IBM WALA[13] and starts by building the call graph of the Android libraries, including all methods as entry points. All call chains containing a `Context.checkPermission(String)` method call are then identified. To find the actual permission string that is used at a `checkPermission` call site, we follow the def-use chain of the string parameter. Once found, we label all callers upstream in the call chain as requiring that permission. Note that different call sites of `checkPermission` will have different permission strings - each such string needs to be propagated correctly upstream, building set of required permissions for each function.

For the `vibrate()` example, the permission enforcement call chain is in Figure 4. The string value of the `vibrate` permission is located by following the def-use chain of the `checkPermission` parameter (dashed lines) until the source `String` constant is found. Once the constant is found, we need to identify which functions on the call chain need this permission. We start by labeling the function that contains the first ("most downstream") call site through which the def-use chain flows. In our case, the def-use chain goes first though the call site of `checkCallingAndSelfPermission` in `IVibratorService.vibrate` (line 4 in Figure 5), where the `VIBRATE` variable is specified as a parameter, so `IVibratorService` is labeled with "`android.permission.VIBRATE`".

After labeling `IVibratorService.vibrate`, the same label must be propagated to callers of that function, but in our case there are no callers except the proxy stub.

Because the communication between Android proxy stubs and their corresponding services (shown as dotted edges in Figure 4) is done through message passing, it does not appear in the actual call chain built by WALA. To work around this problem we add the permissions labels of the service methods to the corresponding proxies; the labels are then propagated to any callers of those methods.

5.2 Taint Rule Generator

The output of the Permission Mapper is a hash table mapping each Android API function that needs permissions to the

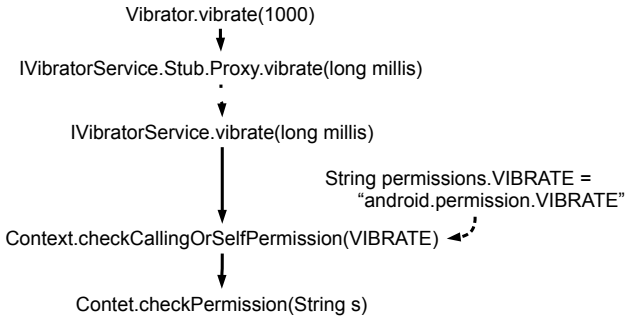

Figure 4: The permission analysis exemplified on the `vibrate()` call

set of one or more permissions that it requires. The taint rule generator turns that information into rules usable by the taint analysis engine.

The rules specify the tracked taint flows: from the sources in the permissions map to the `Activity.setResult` function. The set of rules automatically generated in this way can be manually augmented by providing additional rules describing what application data should be private (protected by custom application permissions).

5.3 Taint Analysis Engine

For the taint analysis engine, we used Andromeda [14], which is highly scalable and precise; it builds on IBM WALA [13]. Andromeda uses as input rules composed of two sets: sources and sinks. The sources are parameters and return values of functions that are the origin of tainted data and the sinks are security-critical methods. The engine tracks data flow from the source through assignments, method calls, and other instructions, until the data reaches a sink method. If the taint analysis engine discovers that tainted data reaches a sink method, the flow is included in the taint analysis engine output.

6. EXPERIMENTAL RESULTS

6.1 Evaluation of the permissions map

We evaluated the Android API Permissions Mapper by directly comparing its output permissions map with that of previous work. The two approaches considered are that of Felt et al. [11], based on automated testing, and that of Bartel et al.[**?**], based on static analysis. To perform the comparison with the work by Felt et al. we eliminated permissions from their map that are enforced through mechanisms other than the `checkPermissions` calls, because our analysis only targets `checkPermissions`-enforced permissions.

Comparing the size of their reference map (which includes 1311 calls that require permissions) with ours (4361 calls with permissions) shows that our tool finds more functions that require permissions. The larger size of our map is partly explained by the lack of false negatives for the analyzed Java APIs. However, a direct comparison is not possible as the input classes on which Felt et al. ran their analysis is not specified in their paper. Our input was the full set of classes in the `android.*` and `com.android.*` packages, as well as Java standard classes that are used inside those (a total of 40,600 methods). Our map also includes functions in internal and anonymous classes, which partially explains the higher number of methods in the permissions map.

Through manual comparison, we identified one false negative in their map (probably due to the automated testing not generating a test and the subsequent analysis not detecting the omission). The function is `MountService.shutdown`, which usually needs the `SHUTDOWN` permission, but if the media is shared it also needs the `MOUNT_UNMOUNT_FILESYSTEMS` permission. The existence of missing permissions in the testing-based methods shows that testing methods, even if party automated and enhanced by manual analysis, cannot offer guarantees with respect to false negatives.

Another reason for the higher number of methods found in our map is the existence of false positives: permissions that are reported as required but are not. We have identified the following sources of false positives, all of which are known weaknesses of static taint analysis:

- Checking for redundant permissions. For example checking for `ACCESS_FINE_LOCATION` or `ACCESS_COARSE_LOCATION` in `TelephonyManager.getCellLocation()`, where either one is sufficient for enforcement; our method reports both, because it is oblivious to control flow.
- Data dependent checks. For example, a check for the `VIBRATE` permissions depends on the value of a parameter such as in `NotificationManager.notify()`.
- Android provides the pair of functions `clearIdentity` and `restoreIdentity` that are used to change all checks so that they are performed on the service instead of the application using the service.

Another advantage of testing-based analysis is that it can cover areas, such as the permissions enforced in native code, which static analysis does not target (such as `RECORD_AUDIO`).

To perform a more detailed comparison of our approach with the work by Felt et al., we compared results obtained for a simple security analysis, identification of overprivileged applications, based on the permission map. This analysis consists of identifying Android applications that request in their manifest more permissions than they actually need to perform their functions. The results are used to reduce the attack surface of applications by removing unused permissions from the manifest. To perform this analysis, we built another static analysis tool based on IBM WALA, which records the API calls that can be performed by an application and computes, based on the permissions map, the permissions required by that application. The set of discovered permissions is then compared to the set of permissions obtained from the application manifest (obtained from the compiled application package through the use of the Android SDK tool `aapt`).

We used both permission maps as input for the analysis of the Top Android Market Free Applications (crawled in December 2011) that were compatible with Android 2.3 and available in the US (354 applications). For a fair comparison, we removed from Felt et al.'s reference map the parts that relates content provider databases to their permissions, as these were outside the scope of our work. After eliminating applications that crashed the `dex2jar` [8] decompiler or generated incorrect bytecode, we were left with 313 applications. Of these, both our analysis and theirs found 116 to be overpriviledged. No permissions identified as unused by us were identified as used by Felt et al., which is consistent with the lack of false negatives expected from a static analysis approach. However, 47 permissions were identified as used by us and as unused by Felt et al..

Comparison with Bartel et al. [**?**] was more difficult as their results are not publicly available. Their analysis focused on Android 2.2, which has a slightly lower number of APIs and so a lower number of permission checks. Their results identify a much smaller number of overprivileged applications: 12% of applications are identified as vulnerable by Bartel et al., but the ratio, as identified by Felt, et al. is closer to 30%. We obtain a 37% rate, which is explained by our not considering native code permissions. Bartel et al. have a similar disadvantage, which should have skewed the rate towards higher values. This inconsistency may have been caused either by their use of a call graph that is too imprecise or by using a different set of applications as input. Our analysis yields a result similar to Felt et al. and uses a similar input (Android Market applications), whereas Bartel et al. used an alternative application store.

Regarding the performance of the tool, the Permissions Mapper runs in under two minutes on our dual core i5 (2.4GHz, 8GB RAM) whereas their map takes two hours to build on a quad core (2.4GHz, 24GB RAM). The performance difference can be explained in part by their trying to eliminate false positives by ignoring permission checks between `clearIdentity` and `restoreIdentity` calls in the kernel; however this technique does not seem to improve accuracy for their experimental results and considerably increases execution time.

6.2 Evaluation of PermissionFlow

We tested PermissionFlow on the same applications used to evaluate the permission map. To confirm the correctness of the results we manually inspected all applications. Out of

the 313 applications, 177 use the `Activity.setResult` with an `Intent` parameter to communicate between components (both internal or external). These 56% of the applications may be vulnerable if they also contain flows from taint sources to sinks and are not configured properly. They can use PermissionFlow to check that they are secure.

To check for correctness, we ran PermissionFlow with our permissions map and the one produced by Felt. Using the map from Felt, PermissionFlow correctly identified two applications as vulnerable and had no false positives. With the permissions map built by our analysis, PermissionFlow outputs a larger set of vulnerable applications, but the additional applications are all false positives. As we saw in the previous section both permissions maps are incomplete: ours does not track permissions enforced through non-Java mechanisms and Felt's allows the possibility of missing permission checks. Choosing one of the two maps amounts to either using a possibly incomplete map (Felt, et al.) and finding no false positives, or identifying the complete set of Java-based flows and accepting some false positives but missing flows based on native code or Linux permissions checks.

Our analysis may have false negatives for applications that pass protected information between components before returning it; for example, `Activity A` may return protected information to `Activity B`, which is improperly secured. We cannot guarantee the identification of such cases because the use of implicit `Intent`s prevents identification of the class names for invoked `Activity`s. For implicit `Intent`s, the receiving class depends on the manifest configuration of all applications installed on the system and may depend on user preferences (if there are multiple `Activity`s with the same `intent-filter` the user is asked to select one that should be invoked). If one requires an analysis without false negatives, our analysis can convert the possible false negatives to possible false positives, by adding, as an additional source for the taint analysis, the `Intent` parameter of the `onActivityResult` callback.

In the following subsections we present the three vulnerable applications discovered by our analysis that leak Android permission-protected information. Section 6.2.4 describes one more vulnerable application that leaks information which should be protected with custom permissions.

6.2.1 Case Study: Adobe Photoshop Express

Adobe Photoshop Express contains an interesting vulnerable flow. The application has an `Activity` that displays a list of `Contacts` and allows the user to pick one. The flow starts from `getContentResolver().query(Contacts.CONTENT_EMAIL_URI)`, and reaches the user interface; from there, the handler for the `Click` operation builds an `Intent` containing the email of the selected contact and returns it to the caller. In Android, read access to the `Contacts` database is protected by the `READ_CONTACTS` permission and callers of this `Activity` work around this restriction.

The `exported` attribute is not set and, because an `intent-filter` is present, the `Activity` is callable from any application. It seems the developer used the `intent-filter` as a security mechanism, which it is not. After disassembling the application and finding the appropriate `category` attribute for the `Intent` and the `Activity` class name, any malevolent developer can exploit it. Even if the user is required to click a contact, there is no way for the user to identify whether the `Activity` returns the information to the legitimate caller (the Photoshop application) or some other application. Because the attack can be performed by any application, the risk in this case is **high**.

6.2.2 Case Study: SoundTracking

The popular SoundTracking application allows users to share a message to their social networks with the name of a song they are listening to and their geo-location. PermissionFlow finds that it is vulnerable to leaking users' geo-location to other applications: the `Activity` responsible is marked with an `intent-filter`, so any other application can invoke it. Manual analysis showed that no fewer than 43 different `Activity`s of this application have `intent-filter`s, and there is no evidence of dynamic or declarative permission checking, suggesting that the developers are confused as to the proper use of the `intent-filter`s. The risk level is **high**.

6.2.3 Case Study: Sygic GPS Application

The Sygic GPS application allows users to take pictures using an `Activity` developed in house, instead of reusing the regular Android camera application. To do this, the `Activity CameraActivity` registers a callback using the `Camera.takePicture` function. The system invokes the callback when the picture is taken and attaches the actual byte array representing the image to it. It then calls `setResult` and `finish`, sending the raw picture to the caller. However, the `Activity` has no `intent-filter`. Because the `exported` attribute is not set, the default value is false. The `Activity` could only be exploited by another application signed by the same developer, so we classify it as **low risk**. For now, none of the other applications of the same developer currently in the market seem to invoke this `Activity`, but this may change in the future. This vulnerability is difficult to detect statically because the source is not in the application code; the application passes a function to the camera API and the operating system calls that function with tainted parameters (the picture array). The `Intent` passes through a message queue, from where it is forwarded to the correct application handler. Identification of this vulnerability was possible because we analyze the application together with the Android libraries and manually added a rule to PermissionFlow that marks the function that distributes the `Intent`s to handlers as having a tainted `Intent` parameter.

6.2.4 Sensitive-information flows

Many applications have access to other types of sensitive information that are not protected by standard Android permissions. For example, a banking application needs to protect credit card information and a social networking application needs to protect family-related and location information. To protect this kind of information, developers should define custom permissions, but because of the coarse-grained nature of the custom permissions (assigned to applications as opposed to APIs) it is not possible to automatically identify the taint sources for such information. For this reason, PermissionsFlow allows specification of additional rules to be used for identifying such vulnerabilities.

Through manual inspection of the market applications we found proof that sensitive information often crosses inter-component boundaries. The existence of these flows shows that PermissionFlow would be a useful tool for developers in configuring their application to not be vulnerable to leaking data protected by either standard or custom permissions.

Some applications fail to protect this information properly. For example, Go Locker has a `lockscreen` passcode selection `Activity` that can leak the phone `lockscreen` password. The Go SMS application from the same developer has a similar flow, but the manifest contains the `sharedUserId` attribute, so it is exploitable. An attacker that can control another application with the same user ID can access the password. Because of the shared user ID restriction, we consider GO SMS as **low risk**.

Some of the inter-component flows of sensitive information that are not protected using standard permissions are described next. These applications are correctly configured, so they are not vulnerable, but their developers could benefit from using our tool because they can easily check if their applications are protected. If the developers took steps to protect sensitive information, these are listed in parentheses after the description of the flows.

- The WeatherChannel application contains an `Activity` that can leak the name of the last recorded video or picture from the camera.
- The Accuweather application can leak the location as selected by the user. Because the string representation of the location does not come from permission protected APIs, the automatically generated rules to not recognize it. However, adding a rules manually is possible and enables detection of this kind of information flow violations.
- Adobe Reader contains an ARFileList `Activity` that returns the absolute path to a file selected by the user (not exported, no `intent-filter`).
- The Facebook application and Facebook Messenger allow the user to select friends from the friends list and returns their profiles, which contain their names, links to their image, etc. (not exported, no `intent-filter`).
- HeyZap Friends has a TwitterLoginActivity that returns information received from the server after login (including username) that may contain information that can be used to compromise the privacy of the users' Twitter account (not exported, no `intent-filter`).
- The mobile CNN application has a similar `Activity` that can leak the user postal code, after he selects it from a list (not exported, no `intent-filter`).
- The Kayak application leaks the user login email if the user logins with the Login screen that the vulnerable application displays (not exported, no `intent-filter`).
- Launcher EX contains an `Activity` that returns the name of an installed application (not exported, no `intent-filter`).
- The GoContactsEX and the FunForMobile applications, contain `Activity`s that return contact information (no `intent-filter`, not exported).
- Google Translate has a flow that involves returning the text of a user selected SMS message (not exported, no `intent-filter`) after the user selects one contact.
- The official Twitter application uses an `Activity` return value to return the Twitter OAuth token that allows access to the twitter API as the user that generated it. The `Activity` is meant to be invoked by other applications (has an `intent-filter`) and is safely configured through declarative permission enforcement. Two other web-related apps, PicsArt Photo Studio and IMDB, contain `Activity`s that return OAuth tokens, but are correctly protected.
- The official Hotmail app contains an `Activity` that returns the URI of an email, but the actual content has to be fetched from a `Content Provider` that declaratively enforces read and write permissions.
- The Walgreens application returns the path and name to the picture taken by the camera. (no `intent-filter`, not exported).

7. DISCUSSION OF FINDINGS

We discovered three vulnerable permissions-leaking applications that may compromise information protected by the `CAMERA`, `READ_CONTACTS` and either `ACCESS_COARSE_LOCATION` or `ACCESS_FINE_LOCATION` permissions.

We also found 70 applications containing flows that need protection but are correctly configured; around 23% of the Android Market applications could use our analysis to confirm proper configuration. As 177 applications had `Activity`s that returned extra information, 56% of the applications could use our tool to pinpoint if they need to secure their protected information against such attacks. PermissionFlow identified all vulnerable applications correctly and had one false positive (because of a bug we are currently investigating); however it reached the 30 minutes time-out on 4 (non-vulnerable) applications.

All the vulnerabilities discovered require the user to perform some action using the GUI of an application that was not explicitly started. This should look highly suspicious to a security-conscious user, making successful attacks less likely to succeed. However, it is easy to perform a similar attack that does not require user intervention and regular users may be easily tricked.

During our manual analysis we paid special attention to applications that handle financial information such as credit card information and online shopping account details and found that these application generally use `Intent`s only for confirmation and not for communication of sensitive information such as PINs, credit card numbers or passwords. Other applications, such as the ones in the case studies above are indeed vulnerable.

We did not find any trace of malevolent attacks performed by the top Android Market applications; most applications

correctly configure internal `Activity`s by not supplying an `exported="true"` attribute or an `intent-filter`.

We believe that part of the cause of the vulnerabilities is the complexity of properly configuring an Android application, as three attributes are involved: `exported`, `intent-filter` and `sharedUid` are not easy to get right.

The complexity involved increases the need for safe defaults. Our recommendation is that Android should require, by default, that any caller of a third-party application must own the permissions required by the callee. This would mean that any `Intent` invocation loses any possible permission collusion capability and only serves as a code reuse mechanism.

For applications such as Barcode Reader that effectively sanitize their data, the developer could add permission attributes to the manifest that lists permissions that should not be needed from callers (whitelist instead of blacklist).

Another helpful but larger change would be for inter- and intra-application `Intent` flows to use separate APIs; this would reduce developer confusion and split a large attack surface into smaller chunks that can be protected each with appropriate tools.

7.1 Recommendations for secure applications

The first and most important advice for security-aware Android developers is to pay close attention to the configuration of their application (specifically, any combination of parameters listed in Table 1 is, without additional checks, vulnerable). If such a parameter combination is needed for functionality reuse or other constraints, here are some ways of maintaining security:

- A safe approach is to always request explicit user confirmation for the invocation of any `Activity` that may be part of an inter-application flow. The user should be informed to which caller the information will be sent. This method has the disadvantage that it decreases the ease-of-use of the application.

- To enforce that callers of your `Activity`s own certain permissions, developers can use either declarative permission requirements in the application manifest and dynamic permission checks using `checkPermission` calls. (both are shown in Section 3)

- Developers should consider using work-arounds for sending sensitive information over inter-component boundaries. For example, several of the applications analyzed leak information from an ordered set of items such as contact names/phone number or zip code. For these applications there is no need for complex mechanisms to avoiding the vulnerability; it may be sufficient to return an integer index to the information database, instead of the actual information; the caller would need to query the database to obtain the actual information.

- Passing sensitive information over inter-component boundaries of the same application in an encrypted form is recommended to protect against unintended callers, but does not help if an attacker has compromised another application with which the current application shares the user id.

8. RELATED WORK

Privilege escalation attacks on Android applications have been previously mentioned in literature [16]. However, such an attack requires usage of native code, careful identification of buffer overflow vulnerabilities and high expertise. We focus only on vulnerabilities specific to Android and help protect the information before such attacks happen.

Michael Grace et al. [17] focused on static analysis of stock Android firmware and identified confused deputy attacks that enable the use of permission-protected capabilities. Our analysis is complementary in that it identifies not actions that are performed, but information that flows to attackers. Also we focus not on stock applications, but third-party applications.

TaintDroid [18] uses dynamic taint tracking to identify information flows that reach network communication sinks. Both PermissionFlow and TaintDroid can potentially support other sinks, and their dynamic approach is complementary to our static approach because it can better handle control flow (for example, paths that are never taken in practice are reported as possible flows by our tool). It can also enforce only safe use of vulnerable applications by denying users the capability to externalize their sensitive information.

SCanDroid [19] is the first static analysis tool for Android and can detect information flow violations. The tool needs to have access to both the vulnerable application and the exploitable application. To the best of our knowledge, SCanDroid is not easily extensible with new taint propagation rules. ComDroid [**?**] is the first tool that analyses inter-application communication in Android. However, it reports warnings which may or may not be vulnerabilities. Automatic rule and permission map generation, as well as analysis of the dangers of using the same mechanism for both intra- and inter-application communications, are contributions that separate our work from theirs.

Mann and Starostin [**?**] propose a wider analysis based on typing rules that can discover flow vulnerabilities. They have no experimental results on applications and did not realize that private components are vulnerable to inter-application attacks. With additional rules, PermissionFlow can analyze all vulnerabilities suggested by them.

A different aspect of the inter-application flow vulnerabilities is described by Claudio Marforio et al. whose work focused on colluding applications[**?**]; they identified several possible covert channels through which malevolent applications can communicate sensitive information, for example by enumerating processes using native code or files. Most of them however are not Android-specific, no tool was built to prevent them, and no vulnerable or malevolent application was found. The contribution of the work consists of identification of the danger of colluding applications for modern permission-based operating systems. Our analysis identifies when permission-protected information leaves the application, but this may be needed to perform the function of the application. Hornyack

et al. [20] propose a system that replaces protected information with shadow data and they concluded that 34% of applications actually need to leak information to perform their function. A combination of PermissionFlow and their tool might lead to better protection of the user privacy, while maintaining functionality.

9. CONCLUSION

This paper proposes a solution for the problem of checking for leaks of permission-protected information; this is an important security problem as such leaks compromise users' privacy.

Unlike previous work, our method is completely automated; it is based on coupling rule-based static taint analysis with automatic generation of rules that specify how permissions can leak to unauthorized applications. We demonstrate the benefits of this analysis on Android, and identify the `Intent` mechanism as a source of permission leaks in this operating system; we found that permissions can leak to other applications even from components that are meant to be private (i.e., accessed only from inside the application) through a more virulent form of confused deputy attack - private activity invocation.

Our automated analysis of popular applications found that 56% of the top 313 Android applications actively use inter-component information flows. Among the tested applications, PermissionFlow found four exploitable vulnerabilities. Because of the large scale usage of these flows, PermissionFlow is a valuable tool for security-aware developers, for security professionals and for privacy-conscious users. Our approach extends beyond Android, to permission-based systems that allow any type of inter-application communication or remote communication (such as Internet access). Most mobile OSes are included in this category and can benefit from the proposed new application of taint analysis. By helping ensure the absence of inter-application permission leaks, we believe that the proposed analysis will be highly beneficial to the Android ecosystem and other mobile platforms that may use similar analyses in the future.

10. REFERENCES

[1] Research2Market, "Android market insights september 2011." `http://www.research2guidance.com/shop/index.php/android-market-insights-september-2011`, 2011.

[2] Apple, "IPhone App Store," Accessed December 2011.

[3] Google, "Android security guide." `http://developer.android.com/guide/topics/security/security.html`, 2012.

[4] Microsoft, "Windows Phone 7 Security Model." `http://www.microsoft.com/download/en/details.aspx?displaylang=en&id=8842`, 2012.

[5] The Linux Foundation, "Meego help: Assigning permissions." `http://wiki.meego.com/Help:Assigning_permissions`, Accessed December 2011.

[6] Nokia, "Symbian v9 security architecture." `http://library.developer.nokia.com/`, 2012.

[7] Passing data between iOS applications, `http://kotikan.com/blog/posts/2011/03/passing-data-between-ios-apps`.

[8] Opensource, "Dex2jar google code repository." `http://code.google.com/p/dex2jar/`, Accessed December 2011.

[9] Opensource, "Android apktool google code repository." `http://code.google.com/p/android-apktool/`.

[10] Opensource, "Java 2 me port of google android."

[11] A. P. Felt, E. Chin, S. Hanna, D. Song, and D. Wagner, "Android permissions demystified," in *Proceedings of the 18th ACM conference on Computer and communications security*, CCS '11, (New York, NY, USA), pp. 627–638, ACM, 2011.

[12] Android Fragmentation information, `http://www.androidfragmentation.com/about`.

[13] T. J. Watson Libraries for Analysis (WALA), `http://wala.sourceforge.net`.

[14] O. Tripp, M. Pistoia, P. Cousot, R. Cousot, and S. Guarnieri, "Andromeda: Accurate and Scalable Security Analysis of Web Applications," in *FASE*, 2013.

[15] D. Sbîrlea, M. Burke, S. Guarnieri, M. Pistoia, and V. Sarkar, "Automatic detection of inter-application permission leaks in Android applications," tech. rep., Rice University, January 2013. `http://www.cs.rice.edu/~vsarkar/PDF/PermissionFlow-TR.pdf`.

[16] L. Davi, A. Dmitrienko, A.-R. Sadeghi, and M. Winandy, "Privilege escalation attacks on Android," in *Proceedings of the 13th international conference on Information security*, ISC'10, (Berlin, Heidelberg), pp. 346–360, Springer-Verlag, 2011.

[17] M. Grace, Y. Zhou, Z. Wang, and X. Jiang, "Systematic detection of capability leaks in stock android smartphones„" in *Proceedings of the 19th Network and Distributed System Security Symposium (NDSS 2012)*, 2012.

[18] W. Enck, P. Gilbert, B.-G. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. N. Sheth, "Taintdroid: an information-flow tracking system for realtime privacy monitoring on smartphones," in *Proceedings of the 9th USENIX conference on Operating systems design and implementation*, OSDI'10, 2010.

[19] A. P. Fuchs, A. Chaudhuri, and J. S. Foster, "SCanDroid: Automated Security Certification of Android Applications," Tech. Rep. CS-TR-4991, Department of Computer Science, University of Maryland, College Park, November 2009.

[20] P. Hornyack, S. Han, J. Jung, S. Schechter, and D. Wetherall, "These aren't the droids you're looking for: retrofitting android to protect data from imperious applications," in *Proceedings of the 18th ACM conference on Computer and communications security*, CCS '11, (New York, NY, USA), pp. 639–652, ACM, 2011.

Biographical Sketches

Dragoş Sbîrlea *Department of Computer Science, Rice University 6100 Main Street, Houston, Texas, 77005 (dragos@rice.edu).*

Mr. Sbirlea is a Ph.D. Candidate in the Habanero Multicore Software Research group at Rice University. He is the author or co-author of 5 scientific publications. His research interests include analysis and optimization of parallel programs. He is a contributor to the Sandia Qthreads library and a member of ACM (Association

for Computing Machinery).

Michael G. Burke *Department of Computer Science, Rice University 6100 Main Street, Houston, Texas, 77005 (mgb2@rice.edu).* Dr. Burke has been a Senior Research Scientist in the Computer Science Department at Rice University since 2009. He received a B.A. degree in philosophy from Yale University in 1973, and M.S. and Ph.D. degrees in computer science from New York University in 1980 and 1983, respectively. He was a Research Staff Member in the Software Technology Department at the IBM Thomas J. Watson Research Center from 1983 to 2009, and a manager there from 1987 to 2000. His research interests include programming models for high performance parallel and big data computing, and static analysis-based tools for: programming language optimization; database applications; mobile security; compiling for parallelism. He is an author or coauthor of more than 50 technical papers and 11 patents. Dr. Burke is an ACM Distinguished Scientist and a recipient of the ACM SIGPLAN Distinguished Service Award. He is a member of IEEE (Institute of Electrical and Electronics Engineers).

Salvatore Guarnieri *IBM Software Group, P.O. Box 218, Yorktown Heights, New York 10598 (sguarni@us.ibm.com).* Mr. Guarnieri is a Software Engineer in the Security Division of IBM Software Group, and a Ph.D. student in the Department of Computer Science of University of Washington. He is an author or co-author of six scientific publications. His research interests include program analysis and security of scripting languages.

Marco Pistoia *IBM Research Division, Thomas J. Watson Research Center, P.O. Box 218, Yorktown Heights, New York 10598 (pistoia@us.ibm.com).* Dr. Pistoia is a Manager and Research Staff Member at the IBM T. J. Watson Research Center, where he leads the Mobile Technologies group. He has worked with IBM for seventeen years. He holds a Ph.D. in Mathematics from New York University, Polytechnic Institute, and a Master of Science and Bachelor of Science degree in Mathematics from the University of Rome, Tor Vergata, Italy. His research interests include program analysis, programming languages, security, and mobile technologies. He is an author or coauthor of ten books, thirty technical papers, fourteen patents and sixty-eight patent applications, and the recipient of two ACM SIGSOFT Distinguished Paper Awards.

Vivek Sarkar *Department of Computer Science, Rice University 6100 Main Street, Houston, Texas, 77005 (vsarkar@rice.edu).* Prof. Sarkar is a Professor of Computer Science and the E.D. Butcher Chair in Engineering at Rice University. He conducts research in multiple aspects of parallel software including programming languages, program analysis, compiler optimizations and runtimes for parallel and high performance computer systems. Prior to joining Rice in July 2007, Vivek was Senior Manager of Programming Technologies at IBM Research. His past projects include the X10 programming language, the Jikes Research Virtual Machine for the Java language, the ASTI optimizer used in IBM's XL Fortran product compilers, the PTRAN automatic parallelization system, and profile-directed partitioning and scheduling of Sisal programs. Vivek became a member of the IBM Academy of Technology in 1995, was inducted as an ACM Fellow in 2008. He holds a B.Tech. degree from the Indian Institute of Technology, Kanpur, an M.S. degree from University of Wisconsin-Madison, and a Ph.D. from Stanford University.